TP9 : Jeu du taquin

Objectif du TP :

- Utiliser la notion de compilation séparée en Ocaml.
- Découvrir et utiliser le module Hashtbl.
- Exploiter l'algorihtme A* et une de ses variantes afin d'orienter une recherche.

Les questions 5, 10, 11, 12, 13 sont obligatoirement à traiter et rendre au format papier pour le 3/01.
Les questions 14 à 20 sont à rendre pour la même date mais sont facultatives.

Cours : partie 1 *Compilation séparée en Ocaml*

Le sujet met à disposition un répertoire TP9_eleves contenant :

- Un module Heap (fichiers heap.ml et heap.mli) permettant la manipulation de files de priorité min.
- Un module Vector (fichiers vector.ml et vector.mli) implémentant des tableaux dynamiques. Ce module sert surtout au bon fonctionnement du module Heap et servira aussi en toute fin de sujet.
- Un code source taquin.ml. Il s'agit de celui qu'il faudra compléter.
- Un Makefile permettant de compiler taquin.ml afin de produire un exécutable taquin que vous exécuterez comme vous le faites en C pour observer le comportement de vos fonctions.

La commande permettant de produire l'exécutable taquin est : make taquin.

1. Utiliser cette commande dans le répertoire contenant tous les fichiers susmentionnés. Observer les fichiers créés et les affichages dans le terminal.

Très grossièrement, les fichiers en .ml sont l'équivalent en C des fichiers en .c et les fichiers en .mli l'équivalent des fichiers en .h. Lors d'une première étape, chacun des trois fichiers en .ml est compilé de sorte à produire un fichier en .cmo contenant le code compilé et un fichier en .cmi qui permet par la suite de lier ensemble les trois codes sources pour fournir l'exécutable taquin.

2. Lancez l'exécutable taquin précédent. Modifier taquin.ml de sorte à ce que l'affichage dans le terminal soit 42. Recompiler et observer l'affichage provoqué par make taquin dans le terminal. Commenter.

Ce comportement est du au fait que make ne recréé pas un fichier s'il est déjà à jour. Pour comprendre le fonctionnement précis de la compilation en Ocaml, vous trouverez plus d'informations sur une partie du processus ici : https://zestedesavoir.com/billets/2081/la-compilation-en-ocaml-avec-ocamlc-et-ocamlopt/

Il faut penser à votre code source taquin.ml un peu comme au code source C qui contiendrait votre main. Comme dans un main en C, il faudra donc tester le bon comportement de vos fonctions en affichant explicitement leurs résultats. Le sujet fournit un certain nombre de fonctions et de tests.

Par ailleurs, votre code source doit être une suite de liaisons entre identifiants et expressions (le plus souvent, l'identifiant est le nom de la fonction et l'expression son code). En particulier, vous ne pouvez pas juste écrire print_string "toto" dans le code source pour afficher toto dans le terminal à l'exécution mais écrire let id = print_string "toto". Comme a priori l'identifiant associé à print_string "toto" (expression de type unit) vous importe peu, vous pouvez le remplacer par le caractère joker _.

Cours : partie 2 *Dictionnaires en Ocaml*

Les dictionnaires en Ocaml sont implémentés via des tables de hachage, les fonctions sur les dictionnaires se trouvent ainsi dans le module Hashtbl. A travers quelques exemples, on présente les principales fonctions utiles de ce module (d'autres existent). Se référer à la documentation pour plus amples détails.

```
(* Création d'un dictionnaire *)
let dico = Hashtbl.create 8;;

(* Ajout de couples clé, valeur *)
Hashtbl.add dico "toto" 4;;
Hashtbl.add dico "42" 2;;
Hashtbl.add dico "rien" 4;;

(* Accès au nombre d'entrées *)
Hashtbl.length dico;;

(* Tests d'appartenance *)
Hashtbl.mem dico "toto";;
Hashtbl.mem dico "titi";;

(* Accès à la valeur associée à une clé *)
Hashtbl.find dico "toto";;
Hashtbl.find dico "zut";;
```

Quelques remarques :

- Hashtbl.create renvoit un objet de type ('a,'b) Hashtbl.t où 'a est le type des clés et 'b le type des valeurs. L'entier passé en argument est une estimation de la taille qu'aura le dictionnaire mais il sera redimensionné si nécessaire : il n'est donc pas problématique de sous-évaluer la taille qu'aura le dictionnaire.
- Hashtbl.mem teste l'appartenance d'une clé dans la table.
- Si une clé n'est pas associée à une valeur dans la table, Hashtbl.find lève l'exception Not_found.

Les exemples suivants illustrent un comportement auquel il faut être attentif de la fonction Hashtbl.add :

```
(* Modification de la valeur associée à une clé *)
Hashtbl.replace dico "toto" 2;;
Hashtbl.find dico "toto";;

(* Le remplacement est un ajout si l'association n'existe pas *)
Hashtbl.replace dico "titi" 2;;
Hashtbl.length dico;;

(* Attention, l'ajout d'une association avec une clé déjà existante...*)
(* ...ne remplace pas mais masque la précédente association *)
Hashtbl.add dico "titi" 42;;
Hashtbl.length dico;;
Hashtbl.find dico "titi";;
```

Comme Hashtbl.replace créé l'association si elle n'existe pas et la modifie sinon, on peut l'utiliser systématiquement à la place de Hashtbl.add pour éviter ce genre de surprises.

TP : partie 0 *Présentation du jeu du taquin*

Le jeu du taquin se joue sur une grille de taille $n \times n$ (classiquement, $n = 4$ et ce sera le cas dans tout le TP) contenant les entiers de $[\![0, n^2 - 2]\!]$. Voici un exemple de configuration initiale :

	0	1	2	3
0	2	3	1	6
1	14	5	8	4
2		12	7	9
3	10	13	11	0

Une configuration fille est obtenue en déplaçant dans la case vide le contenu d'une des cases voisines (en haut, en bas, à droite, à gauche) de cette case vide. Si on déplace par exemple le contenu de la case située à droite de la case libre, c'est-à-dire 12, on obtient la configuration fille suivante :

2	3	1	6
14	5	8	4
12		7	9
10	13	11	0

Le but du jeu du taquin est de parvenir à la configuration finale suivante :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

L'objectif de ce TP est de déterminer une suite de déplacements légaux **de longueur minimale** permettant de passer d'une configuration initiale donnée à la configuration finale. En Ocaml, une configuration sera représentée par le type suivant :

```
type state = {grid : int array array;
              mutable i : int;
              mutable j : int;
              mutable h : int}
```

Plus précisément :

- i et j indiquent les coordonnées de la case libre.
- grid est une matrice de taille $n \times n$, où n est une constante globale préalablement définie, codant le contenu de la grille. Le contenu de la case libre est quelconque.
- h sera la valeur d'une heuristique ultérieurement définie et estimant la distance de la configuration à la configuration finale.

TP : partie 1 *Graphe du taquin*

Une configuration du taquin s'encode naturellement comme une permutation de $[\![0, 15]\!]$ avec 15 représentant la case vide. Le graphe (non orienté) du taquin est défini par :

- ses sommets sont étiquetés par les éléments de $\mathfrak{S}_{16}$,
- il y a une arête de u à v si et seulement si on peut passer de u à v en une étape par l'un des au plus quatre déplacements possibles.

3. Quel est le nombre de sommets de ce graphe ? Son nombre approximatif d'arêtes ? Peut-on raisonnablement le stocker explicitement en mémoire ?

Trouver une suite de déplacements minimale pour résoudre un taquin revient à calculer un plus court chemin dans le graphe du taquin entre la configuration initiale et la configuration finale. Au vu de la taille de ce graphe, on souhaite effectuer cette recherche de manière orientée à l'aide de l'algorithme A*. On se dote dans cette partie de quelques fonctions utilitaires et d'une heuristique permettant de guider A*.

On code un déplacement à l'aide du type suivant : U (up) correspond à un déplacement de la **case vide** vers le haut, D (down) vers le bas, L (left) vers la gauche et R (right) vers la droite.

```
type direction = U | D | L | R | No_move
```

4. Ecrire une fonction possible_moves : state -> direction list qui renvoit la liste des directions de déplacement légales à partir d'une configuration donnée.

Pour chaque configuration c du jeu du taquin, et tout $v \in [\![0, n^2 - 2]\!]$, on note c_v^i la ligne de l'entier v dans la configuration c et c_v^j sa colonne. L'heuristique h que l'on choisit pour orienter la recherche associe à la configuration c l'entier suivant :

$$h(c) = \sum_{v=0}^{n^2-2} \left(|c_v^i - \lfloor v/n \rfloor| + |c_v^j - (v \bmod n)| \right)$$

5. Montrer que cette heuristique est cohérente et admissible. *Indication : que représente $h(c)$?*
6. En s'aidant de la fonction distance fournie, écrire une fonction compute_h : state -> unit qui modifie le champ h de la configuration c prise en entrée de sorte à ce qu'il vaille $h(c)$.
7. Ecrire une fonction delta_h : state -> direction -> int prenant en entrée une configuration c, une direction d et renvoyant la différence $h(c') - h(c)$ où c' est la configuration fille de c lorsqu'on fait le déplacement d. On ne fera que les calculs nécessaires (autrement dit, on ne recalculera pas toute la somme définissant h depuis le début). On pourra se servir de la fonction delta fournie par l'énoncé après avoir compris son fonctionnement.
8. Ecrire une fonction apply : state -> direction -> unit qui modifie une configuration en lui appliquant le déplacement donné par la direction en entrée (qu'on supposera légal sans le vérifier).

Dans cette dernière fonction, on a choisi de modifier la configuration plutôt que d'en recréer une nouvelle suite à un coup. Il sera parfois utile de disposer d'une copie indépendante d'une configuration :

9. Ecrire une fonction copy : state -> state prenant une configuration en entrée et en renvoyant une copie. On pourra utiliser Array.copy en prenant garde au fait que grid est un tableau de tableaux.

TP : partie 2 *Algorithme A**

10. Ecrire une fonction `successors : state -> state list` prenant en entrée une configuration et renvoyant la liste de ses configurations filles. Attention aux partages de mémoire.

Nous avons souvent encodé les arbres associés aux parcours de graphe par un tableau T comme suit :

- $T[i] = i$ si et seulement si i est la racine de l'arbre.
- $T[i] = j \neq i$ si j est le père de i dans l'arbre.

On peut étendre ce principe des tableaux aux dictionnaires : les clés comme les valeurs sont alors des sommets du graphe et la présence de l'association (s, p) dans le dictionnaire signale que le père de s dans l'arbre est p.

11. Ecrire une fonction `reconstruct : ('a,'a) Hashtbl.t -> 'a -> 'a list` qui prend en entrée un dictionnaire encodant un arbre, un noeud x de cet arbre et renvoit le chemin de la racine de l'arbre à x sous forme d'une liste de noeuds.

12. Ecrire une fonction `astar : state -> state list` prenant en entrée un état initial et calculant un chemin de longueur minimale vers l'état final à l'aide de l'algorithme A*. Cette fonction lèvera l'exception `No_path` si aucun chemin n'existe. On utilisera des dictionnaires pour stocker les distances depuis l'origine et les prédecesseurs sur un plus court chemin plutôt que des tableaux.

13. Tester cette fonction sur les exemples fournis. Estimer le temps de calcul nécessaire pour chacun, compter le nombre d'états explorés dans chaque cas, et indiquer la suite de mouvements à effectuer pour gagner dans le cas de `twenty`. On pourra s'aider de nouvelles fonctions qu'on décrira.

TP : partie 3 *Algorithme IDA**

Le facteur limitant dans ce problème est principalement la mémoire au vu de la taille du graphe à explorer. Afin d'améliorer les performances de la partie précédente, on s'intéresse à l'algorithme IDA* qui est un hybride entre l'algorithme A* et le parcours en profondeur itéré, IDS.

Ce dernier repose sur l'algorithme de parcours en profondeur limité à une profondeur maximale suivant :

```
DFS(m, e, p) =
    Si p > m, renvoyer faux
    Si e est le sommet but b, renvoyer vrai
    Pour tout voisin x de e
        Si DFS(m, x, p + 1), renvoyer vrai
    Renvoyer faux
```

Dans cet algorithme, e représente le sommet actuel exploré, p la profondeur actuelle (la longueur du chemin suivi depuis le sommet initial vers celui actuel) et m la profondeur maximale d'exploration autorisée.

14. Montrer que DFS(m, s, 0) renvoit vrai si et seulement si le sommet but est à une distance inférieure à m de la configuration initiale s.

Le parcours en profondeur itéré IDS consiste à effectuer des appels successifs à DFS(0, s, 0), DFS(1, s, 0)... jusqu'à trouver un m pour lequel on obtient une réponse positive, dans l'objectif de déterminer la distance séparant un sommet source s du sommet but b.

15. Déterminer la complexité en temps et en espace d'un parcours en profondeur itéré depuis un sommet initial s situé à distance n du sommet final dans les deux cas suivants :

    a) Le graphe contient exactement un sommet à distance k de s pour tout k.

b) Le graphe contient exactement 2^k sommets à distance k de s pour tout k.

Quel peut être l'intérêt d'effectuer un parcours en profondeur itéré plutôt qu'un parcours en largeur pour déterminer un plus court chemin ?

L'algorithme IDA* est obtenu à partir de IDS en lui ajoutant une heuristique admissible h et en effectuant les modifications suivantes :

- La borne ne concerne plus la profondeur p mais le coût estimé $h(e) + p$.
- Si un parcours avec une borne m a échoué, le parcours suivant se fait avec comme borne la plus petite valeur de $h(e) + p$ qui a dépassé m lors du parcours.

Si l'heuristique est bonne, on va parcourir des fragments d'arbres qui vont croître dans une direction orientée vers le sommet but b. Le pseudo code correspondant pour IDA* suit :

```
IDA*(s) =
    m ← h(s)
    DFS*(m, e, p) =
        c ← p + h(e)
        Si c > m
            minimum ← min(c, minimum)
            Renvoyer faux
        Si e est le sommet but, renvoyer vrai
        Pour tout voisin x de e
            Si DFS*(m, x, p + 1), renvoyer vrai
        Renvoyer faux
    Tant que m ≠ ∞
        minimum ← ∞
        Si DFS*(m, s, 0), renvoyer vrai
        m ← minimum
    Renvoyer faux
```

16. Ecrire une fonction `idastar_length : state -> int option` qui calcule la longueur minimale entre la configuration intiale donnée en entrée et la configuration finale. On ne fera aucun appel à `successors` ; à la place on utilisera une configuration qu'on mutera au fur et à mesure du parcours. Si la fonction détecte que la configuration finale est inaccessible, elle renverra `None`.
17. Montrer que si l'heuristique h est admissible, la fonction `idastar_length` renvoie effectivement la longueur d'un plus court chemin de la source au but s'il existe.
18. Modifier la fonction `idastar_length` en une fonction `idastar : state -> direction Vector.t option` qui renvoit un chemin minimal de coups à effectuer sous la forme d'un `direction Vector.t` pour résoudre le taquin étant donnée une configuration initiale. Elle renverra `None` si un tel chemin n'existe pas.

    On évitera dans `idastar` de revenir immédiatement sur ses pas (autrement dit, on ne testera pas le coup `L` si le dernier coup sur le chemin actuel était `R`). La driection `No_move` peut s'avérer utile.
19. Donner la suite minimale de coups à effectuer pour résoudre le taquin `fifty`.
20. Comparer les performances de `astar` et `idastar` sur le taquin `fifty`.

Ce TP est une reprise d'un sujet de Jean-Baptiste Bianquis, lycée de Parc.